CACHING WITH SELECTIVE MULTICASTING IN A PUBLISH-SUBSCRIBE NETWORK

CROSS REFERENCE TO RELATED APPLICATIONS

The present application incorporates by reference and claims the priority of U.S. Provisional Application No. 60/394,714, entitled "Caching With Selective Multicasting in a Publish-subscribe network," filed July 8, 2002. The present application is also a Continuation-in-Part (CIP) of U.S. Patent Application No. 10/199,356, entitled "Packet Routing Via Payload Inspection," U.S. Patent Application No. 10/199,368, entitled "Method And Apparatus For Content-Based Routing And Filtering At Routers Using Channels," U.S. Patent Application No. 10/199,439, entitled "Method For Sending And Receiving A Boolean Function Over A Network", U.S. Patent Application No. 10/199,369, entitled "Method For Storing Boolean Functions To Enable Evaluation, Modification, Reuse, And Delivery Over A Network," and U.S. Patent Application No. 10/199,388, entitled "Efficient Implementation of Wildcard Matching On Variable-Sized Fields In Connect-Based Routing," all filed July 19, 2002 and all hereby incorporated by reference.

The present application also incorporates by reference the following U.S. Patent Applications, also CIPs of the above-referenced applications, filed March 28, 2003: Application No. 10/400,671, entitled "Method and Apparatus for Reliable Publishing and Subscribing in an Unreliable Network," Application No. 10/400,465, entitled "Method and Apparatus for Content-Based Packet Routing Using Compact Filter Storage and Off-Line Pre-computation," Application No.10/400,453, entitled "Method and Apparatus for Implementing Query-Response Interactions in a Publish-Subscribe Network," Application No. 10/400,462, entitled "Method and Apparatus for Implementing Persistent and Reliable Message Delivery," and, Application No. 10/400,444, entitled "Method and Apparatus for Propagating Content Filters for a Publish-Subscribe Network."

FIELD OF THE INVENTION

The present invention relates to a method and apparatus for caching data in a network core using selective multicasting based upon inspection of a payload in the packet for use in a publish-subscribe network.

BACKGROUND OF THE INVENTION

Network bandwidth is increasing exponentially. However, the network infrastructure (including routers, servers, daemons, protocols, etc.) is still using relatively old technologies. As a result, Internet applications and network routers cannot keep up with the speed of the bandwidth increase. At the same time, more and more devices and applications are becoming network enabled. The load that these devices and applications put on the network nodes have increased tremendously. The increase of network load and number of applications also makes the complexity of implementing and maintaining network applications much higher. As a result, the increase of network bandwidth and the ubiquitous use of network devices and applications can cause problems for routing and transmission of data in the old network infrastructure, particular when publishing content to subscribers.

A model for having networks push information from servers to clients is the publish-subscribe style. In this model, the server becomes a simplified publisher of its information, without regard to which clients may be interested in that information or where they are located in the network. The clients become subscribers for information, with information delivered as it becomes available, potentially without regard to details about where in the network it was published. The network is then responsible for efficiently routing published information to subscribers, for matching information to active subscriptions, and for doing all of this in a way that is transparent to the publishers and subscribers.

Because the complexity of the server is greatly reduced in the publish-subscribe model, the distinction between a heavyweight server and a lightweight client can begin to disappear, or rather to merge into the notion of a peer that can be either publisher, or subscriber, or both. Numerous kinds of applications have a natural affinity for publish-subscribe-style interaction between peers. A common theme underlying many of these applications is that the information being published and subscribed for is in the form of events. For example, an investor buys or sells a stock, causing the price of the stock to change. A traffic incident occurs on a freeway, causing traffic on the freeway to back up. A security hole in a software system is discovered, causing a patch to be developed for the users of the software. A player fires a weapon in an Internet game, causing another player's avatar to die. All of these exemplary phenomena are events that are potentially of interest to large numbers of subscribers and can be propagated over a network to notify those subscribers that the events happened. An event is thus simply a self-contained, succinct piece of information about something potentially interesting that happened at some point in time at some place on the network.

Typically the server or publisher performs the routing decisions for the network in order to instruct the network on where to send published content in the publish-subscribe model. The publisher stores the subscriptions for content that it publishes. Upon receiving or generating new content, the publisher compares the content with each of the subscriptions to identify any matches. If the content (event) satisfies any subscriptions, the publisher pushes the content to the corresponding subscriber via the network. This conventional publish-subscribe model places a tremendous burden on the publishers, particular as more devices become network-enabled and as the number of subscriptions increases.

With greater convergence of untold numbers of applications across the Internet, the possibilities for exploiting event notification become endless. However, those possibilities require a more efficient way to make routing decisions and determine when events satisfy subscriptions, alleviating the burden on the publishers. Thus, a pervasive, persistent event notification service could provide tremendous value-added benefit for Internet applications, as well as other applications and implementations.

SUMMARY OF THE INVENTION

Embodiments of a method and apparatus described herein overcome the disadvantages of the prior art. The embodiments have the advantage of providing redundant and efficient caching of data routed by a publish-subscribe network. Other advantages include enabling a subscriber to retrieve data loss due to a network failure or other error. Advantages also include separating routing from caching functions to provide a more efficient routing.

These and other advantages are achieved, for example, by a method for routing and caching packets of data in a multicast network. The method includes receiving a packet having a header section and a payload section, inspecting the payload section of the packet in a network core for use in determining how to route the packet to subscribers, selectively routing the packet based upon the inspecting, and locally caching data from the packet in the network core. Apparatus including modules for performing these steps are also provided.

These and other advantages are also achieved, for example, by a network for routing and caching packets of data. The network includes an edge routing node that receives and routs packets having a header section and a payload section. The edge routing node includes an intelligent router that routs the received packets and a cache manager. The intelligent router include instructions for inspecting the payload section of the packets in a network core for use in determining how to route the packets to subscribers and selectively routing the packets based upon the inspecting. The cache manager is operatively connected to the intelligent router and includes instructions for locally caching data from the packets in a local cache. The network also includes one or more core routing nodes that receive and rout the packets.

Likewise, these and other advantages are achieved, for example, by an apparatus for routing and caching packets of data in a multicast network. The apparatus includes a plurality of processors and instructions for receiving a packet having a header section and a payload section, inspecting the payload section of the packet in a network core for use in determining how to route the packet to subscribers, selectively routing the packet based upon the inspecting, and locally caching data from the packet in the network core.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are incorporated in and constitute a part of this specification and, together with the description, explain the advantages and principles of the invention.

FIG. 1 is a diagram illustrating intelligent routing in a network core.

FIG. 2 is a network diagram illustrating intelligent routers for publishers and subscribers.

FIG. 3 is a diagram illustrating a network infrastructure for intelligent routers and backbone routers.

FIG. 4 is a diagram of hardware components of an intelligent router.

FIG. 5 is a diagram of publisher and user machines.

FIG. 6 is a diagram of channel managers for intelligent routers.

FIG. 7 is a diagram of software components in a user machine for interfacing the machine with intelligent routers

FIG. 8 is a diagram of software components for an intelligent router.

FIG. 9 is a diagram of a packet structure for a message.

FIG. 10 is a flow chart of a publisher method.

FIG. 11 is a flow chart of a subscriber method.

FIG. 12 is a diagram of channel and subscriber screens.

FIG. 13 is a flow chart of a content-based routing method.

FIG. 14 is a flow chart of a caching method.

FIG. 15 is a diagram illustrating a cache index.

FIG. 16 is a flow chart of an agent method for an outgoing message.

FIG. 17 is a flow chart of an agent method for an incoming message.

FIG. 18 is a diagram illustrating an example of encoding of a message.

FIG. 19 is a diagram of a database structure for storing subscriptions.

FIG. 20 is a flow chart of a wildcard method.

FIG. 21 is a diagram of a cache in a network.

FIG. 22 is a diagram of a backup persistent cache in an upstream router.

FIG. 23 is a diagram illustrating interaction of a cache with a proxy and router.

FIG. 24 is a diagram illustrating cache creation on subscription.

FIG. 25 is a diagram of an indexing tree.

FIG. 26 is a diagram illustrating retrieval from multiple caches.

FIG. 27 is a diagram illustrating interaction of a cache manager with other modules in the system.

FIG. 28 a diagram of a file directory structure for a cache.

DETAILED DESCRIPTION

Overview

An Internet-scale, or other distributed network-scale, event notification system provides applications with a powerful and flexible realization of publish-subscribe networking. In this system, an application program uses event notification application program interfaces (APIs) to publish notifications and/or to subscribe for and receive notifications about events occurring inside the network.

A notification in the system is given a subject, which is a string or other structure that classifies the kind of information the notification encapsulates. Also, a notification is completed with a set of attributes containing information specific to the notification. For example, an application might publish notifications about transactions on the New York Stock Exchange using the subject quotes.nyse and attributes symbol and price. The application might publish an individual notification having specific attribute values, for example with symbol equal to SNE (the stock ticker symbol for Sony Corporation) and price equal to 85.25. Most if not all of the attributes in a notification are predefined, in the sense that they are found in all notifications for the same family of subjects. However, publishers can add discretionary attributes on a per-notification or other basis in order to provide additional event-specific information. Therefore, not all or even any attributes need be predefined.

In this system, subscribers are not restricted to subscribing only for subjects or whole channels. Channels are further explained and defined below. They can include an hierarchical structure specifying, for example, a subject field and one or more levels of related sub-fields (sub-subjects). Thus, subscribers can provide much more finely-tuned expressions of interest by specifying content-based filters over the attributes of notifications. For example, a subscriber might subscribe for all notifications for the subject quotes.nyse having symbol equal to SNE and price greater than 90.00 (indicating perhaps a sell opportunity for a block of shares owned by the subscriber). All notifications matching the subscription can be delivered to the subscriber via a callback or other type of function that the subscriber provides at the time it registers its subscription or at other times. One subscription can be broken down into many filters.

The callback can perform many computations, including something as simple as writing a message to a terminal or sending an e-mail, to something more complex such as initiating the sale of a block of shares, and to something even more complex that initiates new publish-subscribe activity (for example, replacing the existing subscription with a new subscription for a buy opportunity at a price of 75.00, or publishing a new notification that the subscriber's portfolio has been modified).

Applications are aided in their publishing and subscribing activities by agents, for example. The agents can possibly make use of or be implemented with proxies. The agents, when used, provide network connectivity for outgoing notifications and subscriptions and delivery of incoming matching notifications to subscribers. Once a notification enters the network, the system's network of routers propagate the notifications to all subscribers whose subscriptions match the notification. One way of accomplishing this would be to broadcast the notification to all points of the network and then let the application agents decide whether the notification is relevant to their subscribers. However, this is not necessarily a scalable approach—the network would usually be quickly overwhelmed by the load of message traffic, especially in the presence of large numbers of active and verbose publishers. And even if sufficient bandwidth were not a problem, the subscribers would be overwhelmed by having to process so many notifications.

The system's exemplary network is much more efficient in the way it routes notifications. First, it can use multicast routing to ensure that a notification is propagated, for example, at most once over any link in the network. Second, it can employ a large number of sophisticated optimizations on filters to reduce as much as possible the propagation of notifications.

FIG. 1 is a diagram conceptually illustrating this intelligent routing in a network core. A publisher 14 transmits content in messages via an edge router 16 to a network core 10, used in a publish-subscribe network. A publish-subscribe network includes any type of network for routing data or content from publishers to subscribers. The content is transmitted via one or more channels 18 representing logical connections between routers or other devices. An intelligent router 12 in network core 10 determines whether to route or forward the message. In particular, intelligent router 12 can determine if the message includes content as subscribed to by a subscriber 24.

Each subscription encapsulates a subject filter and an attribute filter. Routers can possibly expand a subject filter to the set of matching subjects and merge attribute filters on a per-subject basis. An intelligent router evaluates the subject filter against the subject of notifications, and evaluates the attribute filter against the attribute values in notifications. The syntax for subject filters can possibly use wildcards, and the syntax for attribute filters can use Boolean expressions, both of which are further explained below. The term "filter" is used to describe a set of events that a subscriber is interested in receiving from publishers. Routing rules are generated from the filters and are used by intelligent routers to make routing decisions.

Therefore, if the entire filter set is not satisfied by a message 26, for example, intelligent router 12 drops (discards) message 26, meaning that the message is not forwarded. If any filter of the entire set is satisfied by a message 20 according to the evaluations of subject and attribute filters, for example, intelligent router 12 routes (forwards) message 20 via edge router 22 and possibly other devices to a subscriber 24, or performs other functions internal to router 12 with message 20, according to all the routing and/or action rules prescribed for the matching filter. The search will continue until either the entire set of filters has been exhausted, or decisions about all the rules have been obtained, whichever comes first.

This type of intelligent content-based routing in a network core provides for real-time data delivery of, for example, alerts and updates. Examples of real-time data delivery for alerts include, but are not limited to, the following: stock quotes, traffic, news, travel, weather, fraud detection, security, telematics, factory automation, supply chain management, and network management. Examples of real-time data delivery for updates include, but are not limited to, the following: software updates, anti-virus updates, movie and music delivery, workflow, storage management, and cache consistency. Many other applications are possible for delivery of information for subscriptions.

Table 1 illustrates storing of subscriptions with subjects and predicates for the filtering. They can be stored in any type of data structure, as desired or necessary, anywhere in the network. As explained below, the predicates are components of subscriptions. The subscriptions can be expressed in any way, examples of which are provided below.

Table 1

subscription 1	subject 1	predicate 1
. . .		
subscription N	subject N	predicate N

Table 2 provides an example of a publication and subscription for a quote server. This example is provided for illustrative purposes only, and subscriptions can include any number and types of parameters for any type of data or content.

Table 2

Quote Server Example

Subject Tree
Quotes.NYSE
Quotes.AMEX
Quotes.NASDAQ
	Publication
	subject = Quotes.NYSE
	Attributes
	Symbol = SNE
	Price = 51
	Volume = 1000000

Attributes Symbol Price Volume	Subscription Subject == Quotes.NYSE Filter (Symbol == SNE) & (Price > 55)

The predicates provide the Boolean expressions for the subscription and the subjects provide an indication of a channel for the subscription. Subscriptions can be expressed in many different ways. Use of Boolean expressions is one such example and provides an ability to easily convert the subscription into a subject filter and an attribute filter for content-based routing. Subscriptions can alternatively be expressed without reference to a subject; however, use of a subject or channel (further explained below) provides a context for interpreting and applying filters to attributes.

The routing decisions can be accomplished in the network core and distributed throughout the network, alleviating processing burdens on publisher and subscriber machines, and significantly enhancing the efficiency of the network. FIG. 1 illustrates one publisher, one subscriber, and one intelligent router for illustrative purposes only; implementations can include many publishers, subscribers, and intelligent routers. The term intelligent router refers to a router or other entity having the ability to make routing decisions by inspecting the payload of a packet or message in a network core or other locations.

Network Infrastructure

FIG. 2 is a network diagram illustrating intelligent routers for publishers and subscribers. A routing entity 30 providing channel services is, for example, effectively layered on a network infrastructure, as explained below, for routing messages among intelligent routers. A publisher 32 conceptually includes, for example, an application 34 to receive an indication of published content, such as a pointer for retrieving the content, and an agent 36 to encode the content for network transmission via channel services 30. A collection of logically interconnected intelligent routers 38, 40, 42, 44, 46, and 48 route the content from the publisher using routing rules generated from subject filters and attribute filters for subscriptions. A plurality of links 39, 41, 43, and 45 provide the logical connections between intelligent routers 38, 40, 42, 44, 46, and 48. Other links 37 and 47 provide, respectively, logical connections between publisher 32 and intelligent router 38, and between a subscriber 54 and intelligent router 46. Subscriber 54 includes an agent 50 to detect and receive the subscribed content, and an application 52 to present the content.

A channel can include, for example, a related set of logical multicast connections implemented in a distributed manner. A channel in this exemplary embodiment is a logically related collection of network resources used to serve a community of publishers and subscribers exchanging content. The content is classified according to the channel subject namespace, and the resources are managed, controlled, and provisioned via channel services provided by channel managers. Multiple channels may share the same resources. Channels can provide a highly scalable directory service such as, but not limited to, the following examples: publisher and subscriber information, authentication and authorization information, message types, management information, and accounting and billing information. Channels can also provide, for example, persistence through caching, a fast data delivery mechanism, security, and user and network management. Channels can be used for any other purpose as well.

The filtering by the intelligent routers can occur in a network core to distribute routing decisions. In addition, intelligent routers can also function as edge routers connecting a user device, such as a publisher or subscriber, with the network core. Also, the same device connected to the network can function as both a publisher to push content to subscribers via routing decisions in the network and as a subscriber to received pushed content. The intelligent routers and channels can be connected in any configuration, as necessary or desired for particular implementations, and the configuration shown in FIG. 2 is provided for illustrative purposes only.

FIG. 3 is a diagram of an exemplary network infrastructure for intelligent routers and conventional backbone routers, also illustrating logical connections for channels.

The intelligent routers in this example use existing backbone routers in the network, such as the Internet or other distributed network, and the intelligent routers are thus effectively layered on the backbone routers. In this example, Internet Service Provider (ISP) networks 58, 59, and 60 each include several backbone routers for conventional routing of messages or packets. A plurality of intelligent routers 61-70 are connected with one or more backbone routers in ISP networks 58, 59, and 60. Intelligent routers 61-70 are also interconnected by a plurality of links 73-85, representing examples of links, and can be connected to end user devices by the links as well. Intelligent routers 61-70 can be controlled by one or more administrator machines such as an entity 71, and one or more virtual private network (VPN) controllers such as an entity 72. The ISP networks 58, 59, and 60 would also be connected to publisher and subscriber machines (not shown in FIG. 3). The backbone routers in and among ISPs 58, 59, and 60 are interconnected in any conventional way within the existing network infrastructure.

The intelligent routers 61-70 and links 73-85, as illustrated, can be implemented using existing network infrastructure, and they provide for content-based routing in the network core. The links 73-85 represent logical connections between intelligent routers 61-70 and can be implemented using, for example, existing network infrastructure or other devices. A link, for example, can be implemented using a logical connection called the tunnel. A tunnel includes the hardware, and possibly software, network infrastructure for implementing a link, and one tunnel can be a component of multiple channels. The channels facilitate content-based routing in the intelligent routers by providing logical configurations for particular types of content and thus providing a context for attributes transmitted over the channels. Although intelligent routers can perform routing decisions without channels, the channels enhance the efficiency of content-based routing by the intelligent routers in the network core.

This exemplary embodiment includes use of channels and links. A link is a connection between two routers–albeit intelligent routers. A channel is a network entity encompassing a (typically large) collection of routers, configured statically or dynamically by the interconnecting links to achieve one-to-many or many-to-many logical connections. In particular, a channel is a top-level logical entity describing the essential characteristics of the channel. Under one channel, there could be many subjects. Each subject will form a sub-network (such as a multicast tree) involving a collection of interconnected routers. These subject-based sub-networks can be allocated, oriented, and configured in different manners. The channel, being a collection of all the sub-networks formed for the subjects under it, may resemble a mesh of networks, for example.

FIG. 4 is a diagram of exemplary hardware components of an intelligent router 92, which can correspond with any of the other referenced intelligent routers. A network node 90 can include intelligent router 92 connected with a conventional backbone router 95. Intelligent router 92 includes a processor 93 connected to a memory 94 and a secondary storage 97 (possibly implemented with a detached machine, for example), either of which can store data, as well as cache data, and store applications for execution by processor 93. Secondary storage 97 provides non-volatile storage of data. Under software control as explained below, processor 93 provides instructions to backbone router 95 for it to route (forward) or not route (discard) messages or packets based upon routing rules generated from subject filters and attribute filters for subscriptions. Although shown as implemented in a separate processor-controlled device, intelligent router 92 can alternatively be implemented in an application specific integrated circuit (ASIC) within backbone router 95 to provide the intelligent routing functions in hardware possibly with embedded software. The intelligent routing functions can also be alternatively implemented in a combination of software and hardware in one or multiple routing devices.

FIG. 5 is a diagram of exemplary publisher and subscriber machines. A publisher machine 100 or 118 can include the following components: a memory 102 storing one or more publisher applications 104 and an agent application 105; a secondary storage device 112 providing non-volatile storage of data; an input device 108 for entering information or commands; a processor 114 for executing applications stored in memory 102 or received from other storage devices; an output device 110 for outputting information; and a display device 116 for providing a visual display of information.

A subscriber machine 122 or 140 can include the following components: a memory 124 storing one or more applications 126 and an agent application 128; a secondary storage device 130 providing non-volatile storage of data; an input device 132 for entering information or commands; a processor 134 for executing applications stored in memory 124 or received from other storage devices; an output device 136 for outputting information; and a display device 138 for providing a visual display of information. Publisher and subscriber machines can alternatively include more or fewer components, or different components, in any configuration.

Publisher machines 100 and 118 are connected with subscriber machines 122 and 140 via a network 120 such as the network described above. Network 120 includes intelligent routers for providing distributed routing of data or content in the network core via packets or messages. Although only two publisher and subscriber machines are shown, network 120 can be scaled to include more publisher and subscriber machines. The publisher and subscriber machines can be implemented with any processor-controlled device such as, but not limited to, the following examples: a server; a personal computer; a notebook computer; a personal digital assistant; a telephone; a cellular telephone; a pager; or other devices. Network 120 with intelligent routers can include any wireline or wireless distributed network, connecting wired devices, wireless devices, or both. Network 120 can also potentially use existing or conventional network infrastructure.

FIG. 6 is a diagram illustrating channel managers 150 for intelligent routers. In this example, channel managers 150 are implemented with multiple servers 152, 154, and 156. Each server includes its own local storage 158, 160, and 162. Intelligent routers 164, 166, and 168 contact channel managers for information about particular channels. The channel managers can also provide for data persistence, fail over functions, or other functions. The channel managers thus provide the channel services, which include a database or set of databases anywhere in the network specifying, for example, channel-related information, properties for data persistence, user information for publishers and subscribers, and infrastructure information. The infrastructure information can include, for example, an identification of intelligent routers and corresponding tunnels connecting them, subjects for the channels, and attributes for the channels (a name and type for each attribute). Packets or messages can also carry channel-related information including identification of fixed attributes and variable attributes.

A user when on-line can download channel information. For example, a user can register by using a user name and password. Upon authenticating the user's log-on, the user can open (invoke) a channel and retrieve information about the channel from the channel managers. Publishers can use that information in publishing content, and subscribers can use that information for entering and registering subscriptions.

Channel Managers 152, 154 and 156 preferably form a group to perform the persistent, reliable channel directory service. One of the channel manger will be the primary and the others are backup channel managers. If the primary fails, the neighbor of the primary takes over to be the new primary channel manager to keep the service reliable. Each intelligent router keeps the addresses of these channel managers. If there is one channel managers can not be reached by the intelligent router, it will look for another one to retrieve the information. Devices in the network can use commands, for example, to retrieve channel information, examples of which are provided in Table 3. Intelligent routers can alternatively only have a primary channel manager or more than two channel managers.

FIG. 7 is a diagram of exemplary software components in a stack 180 in a user machine or device for connecting it with a network having intelligent routers. The user machine can be used as a publisher, subscriber, or both, and it can include the exemplary devices identified above. Stack 180 can include one or more user applications 182, which can provide for receiving subscriptions from a user, receiving channel information from a publisher, or receiving content or data to be published. User application 182 can also include any other type of application for execution by a user machine or device.

The stack 180 can also include, for example, an agent 184, an event library 186, a cache library 188, a channel library 190, a messaging library 192, and a dispatcher library 194. Agent 184 provides for establishing network connections or other functions, and Table 3 provides examples of commands implemented by agent 184, which can use proxy commands or other types of commands. Event library 186 logs events concerning a user machine or other events or information. Cache library 188 provides for local caching of data. Channel library 190 stores identifications of channels and information for them. Dispatcher library 194 provides connections with a control path 196, a channel manager 198, and one or more intelligent routers 200, and it can include the exemplary functions identified in Table 4. Messaging library 192 provides a connection with a data path 204.

Tables 5-9 provide examples of messaging APIs in the C programming language. Tables 5 and 6 provide examples of APIs to send and retrieve messages. Tables 7 and 8 provide examples of APIs to send and retrieve notifications. Table 9 provides examples of APIs to send and retrieve control messages. These APIs and other APIs, programs, and data structures in this description are provided only as examples for implementing particular functions or features, and implementations can include any type of APIs or other software entities in any programming language.

Table 3	
Examples of Agent Commands	
command	function
pc.chn.open	open channel, retrieve all information for channel, and locally cache it
pc.chn.close	close channel
pc.chn.getRouterInfo	retrieve information for routers on channel
pc.chn.getAttributeInfo	retrieve information for attributes of channel
pc.chn.getProperties	retrieve properties for channel

Table 4	
Dispatcher Functions	
Server-Side	Listens for connections (sits on accept). Creates a thread to handle each connection. The thread is responsible for receiving and processing all requests coming on that connection.
Client-Side	Creates a thread that initiates a connection and is responsible for receiving and processing all data coming into the connection.

Table 5

Example of API to Send a Message

```
PC_Status    PC_msg_init(ChannelHandle ch, PC_UINT chld, PC_UINT userid,
                 PC_TypeInfo* MsgType, PC_UINT msgTypeSize,
                 PC_msg_SessionHandle *sess);
PC_Status    PC_msg_cleanup(PC_msg_SessionHandle sess);
PC_Status    PC_msg_closeTransport(PC_msg_SessionHandle sess);
PC_Status    PC_msg_create(PC_msg_SessionHandle s, PC_msg_DataType dType,
                 PC_msg_MsgHandle *msg);
PC_Status    PC_msg_delete(PC_msg_MsgHandle msg);
PC_Status    PC_msg_clone(PC_msg_MsgHandle org, PC_msg_MsgHandle *new);
PC_Status    PC_msg_setSubject(PC_msg_MsgHandle msg, PC_CHAR *subject);
PC_Status    PC_msg_setSubjectint(PC_msg_MsgHandle msg,
                 PC_USHORT *subjectArray, PC_UINT arraySize);
PC_Status    PC_msg_setAttrByNameInt(PC_msg_MSGHandle msg,
                 const PC_CHAR *name, PC_INT value); // for each type
PC_Status    PC_msg_setAttrByPosInt(PC_msg_MsgHandle msg,
                 PC_UINT attributePos, PC_INT Value); // for each type
PC_Status    PC_msg_addAttrInt(PC_msg_MsgHandle msg, const PC_CHAR
             *name,
                         PC_INT value); // for each type
PC_Status    PC_msg_send(PC_msg_MsgHandle msg);
```

Table 6

Example of API to Retrieve a Message

```
typedef struct_attribute {
        PC_CHAR                 *name;
        PC_TypeCode             type;
        void                    *value;
        PC_UINT                 arraySize;
} PC_msg_Attribute;
typedef struct_attributeArray {
        PC_UINT                                 size;
        PC_msg_Attribute        **attrs;
} PC_msg_AttributeArray;
PC_Status     PC_msg_init(ChannelHandle ch, PC_UINT chld, PC_UINT userid,
              PC_TypeInfo*
                      MsgType, PC_INT msgTypeSize, PC_msg_SessionHandle
              *sess);
PC_Status     PC_msg_cleanup(PC_msg_SessionHandle sess);
PC_Status     PC_msg_recv(PC_msg_SessionHandle sh, PC_msg_MsgHandle *msg);
PC_Status     PC_msg_ctrlRecv(PC_msg_SessionHandle sh, PC_msg_MsgHandle
              *msg);
PC_Status     PC_msg_getSequenceNum(PC_msg_MsgHandle msg, PC_UINT
              *seqNo);
PC_Status     PC_msg_getPublisherInfo(PC_msg_MsgHandle msg,
              PC_msg_PublicInfo *pub);
PC_Status     PC_msg_getSubject(PC_msg_MsgHandle msg, PC_CHAR **subject);
PC_Status     PC_msg_getSubjectInt(PC_msg_MsgHandle msg,
                      PC_USHORT **subjectArray, PC_INT *size);
PC_Status     PC_msg_getDataType(PC_msg_MsgHandle hMsg,
                      PC_msg_DataType *dataType);
PC_Status     PC_msg_getAttrByPosInt(PC_msg_MsgHandle msg,
              PC_UINT pos, PC_INT *val); // for each type
PC_Status     PC_msg_getAttrValueByNameInt(PC_msg_MsgHandle msg,
                      const PC_CHAR *name, PC_INT *val);
PC_Status     PC_msg_getAttrTypes(PC_msg_MsgHandle msg, PC_TypeCode* Types,
                      PC_INT *arraySize);
PC_Status     PC_msg_getAttributeByPos(PC_msg_MsgHandle msg,
                      PC_UINT attributePos, PC_msg_Attribute **attr);
PC_Status     PC_msg_getAttributeByName(PC_msg_MsgHandle msg,
                      const PC_CHAR *name, PC_msg_Attribute **attr);
PC_Status     PC_msg_getPredefinedAttributes(PC_msg_MsgHandle msg,
                      PC_msg_AttributeArray **attrs );
PC_Status     PC_msg_getDiscretionaryAttributes(PC_msg_MsgHandle msg,
                      PC_msg_AttributeArray **attrs);
Void          PC_msg_freeAttribute(PC_msgAttribute *attr);
Void          PC_msg_freeAttributeArray(PC_msg_AttributeArray*attrArray);
```

Table 7

Example of API to Send a Notification

```
ChannelHandle ch;

PC_msg_MsgHandle msg;
PC_msg_SessionHandle sh;
PC_msg_TypeInfo     Types[2];
Types [0].type = PC_STRING_TYPE;
Types [0].name = “company”
Types [1].type = PC_INT_TYPE;
Types [1].name = “stockvalue”

PC_msg_init(ch, chId, userId, Types, 2, &sh)

PC_msg_create(sh, PC_MSG_DATA, &msg);
PC_msg_setAttrValueByNameInt(msg, “stockvalue”, 100);
PC_msg_setAttrValueByPosString(msg, 1, “PreCache”);
PC_msg_addAttrString(msg, “comment”, “mycomments”);

PC_msg_send(msg);
PC_msg_delete(msg);
PC_msg_closeTransport(sh);
        PC_msg_cleanup(sh);
```

Table 8

Example of API to Retrieve a Notification

```
ChannelHandle ch;

PC_msg_MsgHandle msg:
PC_msg_SessionHandle sh;
PC_msg_TypeInfo     Types[2];
PC_msg_AttributeArray *attrArray;
PC_CHAR *company;
PC_INT value;
Types [0].type = PC_STRING_TYPE;
Types [0].name = “company”
Types [1].type = PC_INT_TYPE;
Types [1].name = “stockvalue”

PC_msg_init(ch, chId, userId, Types, 2, &sh);
While (1) {

        PC_msg_recv(sh, &msg);
        PC_msg_getAttrValueByPosString(msg, 0, &company);
        PC_msg_getAttrValueByNameInt(msg, “stockvalue”, &value);
        PC_msg_getDynamicAttributes(msg, &attrArray);
        PC_msg_freeAttributeArray(attrArray);
        PC_msg_delete(msg);
}
PC_msg_closeTransport(sh);
        PC_msg_cleanup(sh);
```

Table 9

Example of APIs to Send and Retrieve Control Messages

Sender Side Code	Receiver Side Code

ChannelHandle ch;
PC_msg_MsgHandle mh;
Int chld = 10;
// Get a Channel handle for channel 10
PC_msg_init(ch, chld, publd, NULL, 0,
&sh)
PC_msg_create(th,
PC_MSG_CONTROL,
                &mh);
PC_msg_setSubject(mh,
"#.ADD_SUBJECT");
PC_msg_addAttrInt(mh,,"Channelld",
                chld);
PC_msg_addAttrString(mh,
                "Subject", "Quote.cboe");
PC_msg_send(mh);
PC_msg_delete(mh);
	ChannelHandle ch;
	PC_msg_MsgHandle msg;
	PC_msg_init(ch, chld, subld, NULL, 0, &sh);
	
	for (;;) {
	        PC_msg_recv(sh, &msg);
	        PC_msg_getSubject(msg, &subject);
	        PC_msg_getAttrValueByNameInt(
	                msg, "Channelld, &chld);
	        PC_msg_getAttrValueByNameString(
	                msg, "Subject", &subject);
	        PC_msg_delete(msg);
	}
	PC_msg_closeTransport(sh);
	PC_msg_cleanup(sh);

FIG. 8 is a diagram of exemplary software components 210 for an intelligent router such as those identified above and intelligent router 92 shown in FIG. 4. Software components 210 can be stored in, for example, memory 94 for execution by processor 93 in intelligent router 92. Components 210 include, for example, a filtering daemon 212, a dispatcher 214, a routing daemon 216, and a cache manager 218. Filtering daemon 212 provides filtering for content-based routing to process content for subscriptions according to routing rules, as explained below. Dispatcher 214 provides for communication of control messages such as those required for propagating filters via path 220, and the dispatcher can also provide for a single point of entry for users and one secure socket with channel managers, enhancing security of the network. In other words, users do not directly contact channel managers in this example, although they may in alternative implementations. Dispatcher 214 uses control messages to obtain attributes (name-value pairs) from a channel manager.

Routing daemon 216 provides for communication with a data path 222, which can occur via a conventional backbone router as illustrated in FIG. 4 or other routing device. Cache manager 218 provides for local caching of data at the network node including the corresponding intelligent router. The operation of cache manager 218 is further explained below, and it provides for distributed caching of data throughout the network core.

Content-based routing can be implemented at the kernel level, as an alternative to the application level. Memory accessible by the kernel is separate from that in the application layer. To have content-based routing running in the application requires, for example, that message data be copied from the kernel memory area to the application area, and switching the context of the application from that of the kernel to that of the routing application. Both can induce substantial overhead. If instead the kernel is modified to support content-based routing, the routing could take place much faster being rid of the overhead described above.

With this feature of content-based routing in the kernel, the routing daemon 216 may or may not directly send or receive data via the data path 222, depending on the implementation. The daemon is a process running in the application layer, pre-computing the content-based routing table to be injected into the kernel. Once injected, however, the routing table can be used by the kernel to make routing decisions. Similarly, the filtering daemon pre-computes the filtering table and injects it into the kernel. In this kernel implementation, neither the routing daemon nor the filtering daemon would directly interact with the data path.

FIG. 9 is a diagram of an example of a packet structure 230 for a message possibly including content for subscriptions. A packet or message for use in content-based routing includes, for example, a header section and a payload section. The header section specifies routing or other information. The payload section specifies data or content, or an indication of the data or content. Packet structure 230 includes an IP header 232, a User Datagram Protocol (UDP) Transmission Control Protocol (TCP) header 234, a length value 238, one or more subject fields 240, and one or more attributes 242. Packet structure 230 illustrates a basic structure for a length value and the subjects and attributes. A packet used in content-based routing can also include other or different elements, such as those illustrated in the example of FIG. 18 explained below, and packets for content-based routing can be configured in any manner. Also, the attributes can include discretionary attributes appended to the end of a message, for example.

These discretionary attributes are ad-hoc information, for example, added by the publisher (or even routers) that cannot necessarily be conveyed using the message format prescribed for the channel.

Publisher and Subscriber Methodologies

FIG. 10 is a flow chart of an exemplary publisher method 250 for use by a publisher to set-up a channel and publish content. Method 250 can be implemented, for example, in software modules including agent 106 for execution by processor 114 in publisher machine 100. In method 150, agent 106 in the publisher machine receives a publisher creation of a proxy for a channel (step 252). The proxy provides for communication with the network. Agent 106 determines a message format for the channel through an interface (step 253), and the format information can be obtained from, for example, the channel managers or other entities in the network. Agent 106 sets up the proxy for the channel using the received channel information (step 254), which includes receiving attributes for the channel (step 256) and creating a notification on the channel (step 258). The notification provides content for devices "listening" for content on the channel. The attributes define parameters and characteristics for the notification.

Agent 106 transmits an identifier (ID) of the channel and content information to intelligent·routers in the network core or elsewhere for use in processing subscriptions (step 260). The publisher populates the notification attributes with appropriate values (step 261), and the publisher can then publish content on notification in accordance with the channel attributes (step 262). Steps 260-262 in this example accomplish publishing the notification, which can alternatively involve different or additional steps depending upon a particular implementation. Therefore, the information associated with a notification in this example is partitioned into an ordered sequence of attributes, each of which has a name, a position within the notification (starting at 1), a type, and a value. Alternatively, attributes can have different characteristics depending upon a particular implementation. Attributes can include, for example, predefined attributes, discretionary attributes, or both.

The intelligent routers can use the channel ID in a packet to obtain the attributes for the corresponding channel, which determines the structure or format for packets transmitted via the channel. In particular, each packet can contain, for example, a tag associated with a channel ID and other header information such as a publisher ID and subjects. The tags can be used to map subjects to numbers in the message format, an example of which is shown in FIG. 18. Small integer values, for example sixteen bit values, can be used for the numbers. Alternatively, any other type of numbers or information can be used to map the subjects. Mapping subjects to numbers can provide particular advantages; for example, it can save space in the message format and provide a uniform or standard way to specify indications of the subjects in the message so that they can be quickly located and identified. Intelligent routers can locally store the mapping or, alternatively, use the numbers to remotely obtain the corresponding subject through a command.

Table 10 illustrates a structure for mapping numbers to subjects, in this example using integer values. The subject tree parameter in the table indicates that a subject can include one or more subject fields in an hierarchical relationship; for example, a subject tree can include a string of subject fields demarcated by particular symbols. Examples of subject trees are provided in Table 2. As an example, a subject tree quotes.nyse includes a subject "quotes" and a sub-field "nyse" with those two terms demarcates by a "." as found in URLs or other network addresses. Aside from using periods and specifying URL-type strings, subject trees can be specified in any way using any characters and symbols for demarcation.

Table 10

Number	Subject Tree
integer value 1	subject tree 1
integer value 2	subject tree 2
. . .	
integer value N	subject tree N

Thus, knowing the packet format or structure for a particular channel, the intelligent routers can quickly locate subjects and attributes, or other information, in the packet for content-based routing. For example, a channel can specify byte positions of subjects and attributes transmitted over the channel, making them easy to locate by counting bytes in the packet. Alternatively, intelligent routers can parse packets to locate subjects and attributes, or other information.

Table 11 provides an example of a publisher program in the C++ programming language. Table 12 provides an example of an API to create a channel. Table 13 provides an example of a channel configuration file maintained by a channel manager (see FIG. 6) and providing channel-related information, as illustrated. The system can alternatively have a global channel manager providing IP addresses of geographically dispersed servers functioning as local channel managers in order to distribute the processing load.

Table 11

Example of Publisher Program

```
#include “PC_evn_Notification.h”
#include “PC_evn_Proxy.h”

using namespace precache::event;

int main(int argc, char argv[])
{
        PC_UINT QuotesRUs = myChannelofInterest; // channel ID
        PC_UINT myID = myPublisherID; // publisher ID

        try {
                Proxy  p(QuotesRUs, myID);
                Notification n1(p, “quotes.nyse”);
                n1.SetPredefinedAttr(“symbol”, “LUS”);
                n1.SetPredefinedAttr(price”, 95.73);
                p.Publish(n1);

                Notification n2(p, “quotes.nyse”);
                n2.SetPredefinedAttr(1, “SNE”);       // attribute symbol is in position 1
                n2.SetPredefinedAttr(2, 80.18);       // attribute price is in position 2
                p.Publish(n2);
        }
        catch (InvalidChannelException icex) {
                cerr << “bad channel” << endl;
        }
        catch InvalidSubjectException isex) {
        }
        catch (InvalidNotificationException inex) {
                cerr << “bad notification” << endl;
        }
        catch (Exception ex) {
                cerr << “unknown error” << endl;
        }
}
```

Table 12

Example of API to Create a Channel

```
PC_Status rc;

rc = PC_chn_create(Provider_info, authinfo, ConfigurationFile, &hChannel);

/* the first one primary channel manager */
rc = PC_chn_addChannelManager (hChannel, “10.0.1.1”);

/* secondary channel manager */
rc = PC_chn_addChannelManager (hChannel, “10.0.2.2”);

*/
rc = PC_chn_setProperties (hChannel, ConfigurationFile);

/*
Set the message type (only in fixed part of the message)
by using rc = PC_chn_setAttributeType(hChannel, name, position, attributeType).
The type information is propagated to all edge routers.
*/
rc = PC_chn_setAttributeType(hChannel,”Priority”,1,PC_UINT 16_TYPE);
rc = PC_chn_setAttributeType(hChannel,”Alarm_Name”,2, PC_STRING_TYPE);
rc = PC_chn_setAttributeType(hChannel,”Alarm_Time”,3, PC_INT32_TYPE);
rc = PC_chn_updateAttribute(hChannel);
rc = PC_chn_close(hChannel); /* finish channel creation */
```

Table 13

Example of a Channel Configuration File

```
# Channel Setup - Read by Channel API, event and messaging
# Each channel entry information is tagged with the
# type of information e.g.,
#  [ChannelComm 5] for Channel 5 Communication related information
#  [ChannelSubjects 5] for subject related information in channel 5
#  [ChannelAttributes 5] for attribute information in channel 5
#
# The Channel id is appended to the tag to indicate
# the channel that the information belongs to
# e.g.,  [ChannelComm 5] indicates routing information
# for channel 5.
#
# All the fields need not be set.  For example if
# running with the central server, the MulticastIP is
# not needed.

[ChannelComm 5]
```

```
MulticastIP=225.0.0.1
RouterIP=test3
RouterPort=12345
ProxyPort=9015
ProxyCtrlPort=9016

[ChannelSubjects 5]
NumberOfSubjects=2
subject1= #.SUBSCRIPTION
mapping1=0.100
subject2=Quotes.Nyse
mapping2=102.101

[ChannelAttributes 5]
NumberOfAttributes=4
name1=StockId
type1=PC_UINT_TYPE
name2=Company
type2=PC_CHARARRAY_TYPE
name3=Price
type3=PC_FLOAT_TYPE
name4=Volume
type4=PC_UINT_TYPE
```

FIG. 11 is a flow chart of a subscriber method 264 for use in receiving and processing subscriptions. Method 266 can be implemented, for example, in software modules including agent 128 for execution by processor 134 in subscriber machine 122. In method 264, a graphical user interface (GUI), for example, presents an indication of available channels to a user (step 266), which can be accomplished by application 126. The information identifying the channels can be received from, for example, the channel managers providing channel-related information. Any type of application 126 can be used for presenting identifications of channels in any particular way or format. The application receives a user's selection of a channel (step 268) and calls an API or other program for the selected channel (step 270). The API presents subscription options to the user for the channel corresponding with the selected option (step 272). The API receives values for the subscription from the user (step 274) and sends the subscription to agent 128 for processing, as explained below (step 276).

The parameters for the subscription can include, for example, the predicates as illustrated in Table 1. Each channel can use its own API, for example, in order to process subscriptions according to the particular requirements or parameters for the corresponding channel. These APIs can include, for example, web-based or Java-based APIs for receiving subscriptions and can use any type of user interface and processing to receive information for a subscription and pass it along to the agent application.

FIG. 12 is a diagram conceptually illustrating channel and subscriber screens or GUIs 278 and 284, which can be used in conjunction with method 264 for receiving a subscription. Screen 278 includes a plurality of sections 282 identifying available channels for selection by a user. Upon selection of a particular channel, screen 284 can be displayed for receiving a user's values for the subscription in a section 286. A user can select a section 288 to submit the subscription or select a section 290 to cancel the subscription. Screens 278 and 284 can be formatted as, for example, HyperText Markup Language (HTML) web pages or in any other format. Also, the screens can include any configuration of sections and content, possibly including, for example, text, graphics, pictures, various colors, or multi-media information in order to provide, as desired, a user-friendly and visually appealing interface for subscribers. The screens can also include a toolbar 280 providing, for example, conventional browser functions.

Table 14 provides an example of a subscriber program in the C++ programming language.

Table 14

Example of Subscriber Program

```
#include <unistd.h>
#include <iostream>
#include “PC_evn_Filter.h”
#include “PC_evn_Subscription.h”
#include “PC_evn_Proxy.h”

using namespace precache::event;

class SubscriberApp : public Subscriber
{
private”:
        PC_UINT notificationCount = 0;

public:
        SubscriberApp() {}  // default constructor

        void run()
        {
                PC_UINT QuotesRUs = myChannelofInterest; // channel ID
        PC_UINT myID = myPublisherID;               // publisher ID

        try {
                Proxy           p(QuotesRUs, myID);
                FilterFactory*  factory = FilterFactory::GetFilterFactory();
                Filter*         f = factory->CreateFilter(p, “symbol == \”LU\””);
                PC_INT          c1 = 0;

                SubscriptionHandle sh = p.Subscribe(“quotes.nyse”, f, this,
                                                    (void*)&c1);

                while (notificationCount < 2) {        // let notify() get some
                                                       // notifications
                        sleep(5);
                }
                p.Unsubscribe(sh);
        }
        catch (InvalidChannelException icex) {
                cerr << “bad channel”<< endl;
        }
        catch (InvalidSubjectException isex) {
                cerr << “bad subject” << endl;
        }
        catch (InvalidChannelException ifex) {
                cerr << “bad filter”<< endl;
        }
        catch (InvalidSubscriptionHandleException ishex) {
                cerr << “bas subscription handle” << endl;
        }
        catch (Exception ex) {
                cerr << “unknown error” << endl;
        }
    }
    void Notify(Notification* n, void* c)          // this is the callback method
    {
        if (*(PC_INT*)c == 0){ // check the closure object
                PC_STRING  symbol;
                PC_FLOAT   price;

                n->GetPredefinedAttr(“symbol”, symbol);
                n->GetPredefinedAttr(“price”, price);
                cout << “The price of “ << symbol << “ is ” << price << endl;
                notificationCount++;
        }
    }
};

int main(int argc, char argv[])
{
    SubscriberApp a;
```

```
        a.run();
        }
```

Content-Based Routing Via Payload Inspection and Channels

FIG. 13 is a flow chart of a content-based routing via payload inspection method 300. Method 300 can be implemented, for example, in software modules for execution by processor 93 in intelligent router 92, as represented by filtering daemon 212. Alternatively, it can be implemented in an ASIC or a combination of hardware and software. The content-based routing as illustrated in method 300 can be performed in intelligent routers anywhere in the network, such as in the network core or in edge routers.

In a general sense, the content-based routing involves inspecting a payload section of a packet in order to determine how to process the packet. This content-based routing methodology can include, for example, processing a list of subscriptions (using filters, for example) in any order, comparing a message subject-by-subject and attribute-by-attribute with routing rules to determine a routing for the message, and performing the processing in a network core. The rules can include rules governing in-router processing or any rules associated with a filter. These routing decisions can thus be distributed throughout a network core. The use of subjects as represented by channels determines a message format, thus providing an intelligent router with a way of quickly locating attributes within the message, for example by knowing their byte positions in the message or packet for a particular channel.

In method 300, intelligent router 92 receives a packet for a message (step 302). It determines from the packet a channel ID for the corresponding message (step 304) and retrieves attributes for the channel using the channel ID (step 306). In this example, the type of channel (determined from the channel ID) determines locations and data types of attributes in the packet. The attributes for the channel can be locally stored or retrieved remotely such as via a channel manager. Intelligent router 92 retrieves a filter, which corresponds with a subscription (step 308). The filter includes one or more attribute tests, usually a group of attribute tests for subscriptions. Intelligent router 92 applies attributes in the packet to the corresponding attribute test(s) in the filter description (step 310).

If all the attribute test(s) in the filter description produce a positive result (step 312), meaning the attributes satisfy all the attribute test(s), the intelligent router executes a set of functions prescribed by the rules associated with the filter (step 314). These functions can include, for example, routing the packet to the next link, and/or performing some action or computation with the content of the packet at the local router as prescribed by the rule(s). The action or next link can be identified, for example, in a data structure specifying the corresponding subscription. When the rule is a link, it typically identifies the next network node to receive the packet, which can include an intelligent router, backbone router, a network-connected device, or other entity. Alternatively, the next links can be specified or associated with the subscriptions in other ways.

If all the attribute test(s) in the filter description did not produce a positive result (step 312), meaning the attributes do not satisfy all the attribute test(s), the filter is declared a mismatch (step 315). The intelligent router recursively follows the above procedure until all the attribute tests in the filter description are exhausted or a first negative result is encountered, whichever comes first.

Once all the attribute tests have been processed for this filter, the intelligent router determines if more filters exist (step 316) and, if so, it returns to step 308 to retrieve the attribute test(s) for the next filter to process the attributes for it. The matching procedure (steps 308, 310, 312, 314, 315, and 316) continues until either the complete set of filters is exhausted, or results for all the action or routing rules can be determined, whichever comes first. If the packet does not satisfy any filter, it will be dropped (discarded) and not forwarded.

Intelligent router 92 can sequence through the filters in any particular order. For example, as illustrated in Table 15, intelligent router can store the filters for subscriptions in a file or routing table and linearly sequence through them to apply the attributes to filters (attribute tests). Alternatively, the routing table can include links or pointers to the filters.

The content-based routing can optionally use more than one method at the same time, depending on the applications and performance-enhancing heuristics such as the switching of algorithms based on traffic conditions, for example. The filters for the processing can optionally be encrypted, decrypted, transformed, and merged at a router in the network for use in performing inspecting of a payload section for the content-based routing. For example, a subscription such as price > $3.54122 may be truncated to price > $3.54 because the publications in the application are known not to contain currency attributes beyond the second decimal points. Also, foreign currency may be translated into U.S. currencies as well when a publication sent from overseas reaches the first router located in the U.S., for example.

As an alternative to a linear approach, intelligent router 92 can select filters for processing in other orders or according to various algorithms that can possibly enhance the speed and efficiency of processing. Table 16 provides examples of subscriptions and corresponding links for them; in these examples, the subjects relate to a particular channel and the subscriptions for the subjects can be represented by routing rules for the filters. The subjects can include, for example, network addresses such as Uniform Resource Locators (URLs) identifying a source of content.

Table 15	
Channel 1	
Subscriptions	Links
filter 1a	links 1a
filter 2a	links 2a
. . .	. . .
filter Na	links na
. . .	
Channel N	
Subscriptions	Links
filter 1N	links 1a
filter 2N	links 1b
. . .	. . .
filter NN	links 1n

Table 16

Content Predicate	Links
sub = “quote.optimist” & (($1 > 5 & $2 = “LU”) | ($1 > 30 & $2 = “T”))	xl0, xl1
(sub = “sony.music” | sub = “sony.movie”) & $1 > 30 & $4 = “Beethoven”	xl1, xl3
sub = “movie.ratings” & ($1 > 1999 | $2 = “Kurosawa”) & $3 = “**”	xl1, sl5

Caching at Network Nodes

FIG. 14 is a flow chart of a caching method 320. Method 320 can be implemented, for example, in software modules for execution by processor 93 in intelligent router 92, as represented by cache manager 218. Alternatively, it can be implemented in an ASIC or a combination of hardware and software, either in the same or different physical device as the corresponding intelligent router. In method 320, intelligent router 92 receives a message having data or content, a channel ID, and subjects (step 322). Intelligent router 92 time marks the data (step 324) and locally caches it such as in memory 94 or secondary storage 97 (step 326). It indexes the cached data by, for example, channel ID, subjects, and time stamps (step 328).

If intelligent router 92 receives a request for data (step 330), it retrieves cached data, using the index, according to the request (step 332). Intelligent router 92 transfers the cached data to backbone router 95 or other routing entity for eventual transmission to the requestor or others. Method 320 can be repeatedly executed in order to continually cache data and retrieve cache data in response to requests.

FIG. 15 is a diagram illustrating a cache index (336) for use with method 320. Cache index (336) receives data (338) and stores it with time stamps (340). As data is gathered, it is marked upon every duration of delta t, where delta t represents the time between marks, for example t_2 - t_1. Other types of indexes for time marking in any way can alternatively be used.

Table 17 conceptually illustrates indexing of cached data. Table 18 conceptually illustrates a data structure for storing a connection history for caching. Table 19 provides examples of data structures for use in locally caching data in network nodes having intelligent routers.

The time marking can occur at any fixed or variable interval. For example, data can be cached and indexed every five minutes. Upon receiving a command to retrieve cached data (such as #.getCache) specifying a time and subject, channel manager 218 uses the cache index to determine if it can retrieve cached data corresponding with the request for step 332.

Each subject or channel can include, for example, its own IP address in a multicast tree and a set of intelligent routers. Therefore, Table 18 represents a connection history among such routers that can be locally stored a user machine; if an edge router fails, the machine can access the connection history to determine how to reconnect with upstream routers for the channel when the edge router comes back on-line. It can also execute a get cache command for the duration of the time that it was disconnected in order to obtain any pending content for subscriptions, for example.

Table 17			
t_1	channel ID 1	subjects 1-n	pointer 1 to cached data
t_2	channel ID 2	subjects 1-n	pointer 2 to cached data
t_n	channel ID N	subjects 1-n	pointer N to cached data

Table 18			
Connection History			
time	router	network addresses	
t_1	R2	UR2	UR3
t_2	R2	UR2	UR3
. . .			

Table 19
Examples of Cache Data Structures for Intelligent Router
Channel Node

```
Struct ChannelNode {
        PC_UINT                 unChanld;
        PC_AttributeInfo        *pAttrinfo;
        PC_BOOL                 bPersistent; /* Persistent or RT*/
        PC_UINT                 unTimeout;
        PC_UINT                 unTimeGranularity;/* in minutes */
        PC_INT                  nDirFd;
        HashTable               *pFirstLevelSubjs;
        }
```

Subject Node

```
Struct SubjectNode {
        PC_USHORT               unSubjectld;
        PC_UINT                 unSubjLevel;
        Void                    pParent;        /* Channel or Subject */
        PC_INT                  nDirFd;
        HashTable               *pNextLevelSubjs;
        DataNode                *pData;
}
```

Data Node

```
Struct DataNode {
        PC_INT                  nDirFd;
        SubjectNode             *pParent;
        LastTimeGrainNode       *pLastTGrainData;
        DLIST                   *pStoredData;/*list StoredTimeGrainNode */
        PC_Mutex                mStoredDataLock;
        }
```

Stored Time Grain Node

```
Struct StoredTimeGrainNode {
        PC_UINT                 unStartTime; /* in minutes */Chanld;
        PC_UINT                 unEndTime; /* in minutes */
        PC_INT                  nFd;
}
```

Last Time Grain Node

```
Struct LastTimeGrainNode {
        PC_CHAR                 pLastTGrainData;        /* could be a list */
        PC_UINT                 unLastTGrainStartTime;
        PC_BOOL                 bReadyToStore;
        PC_Mutex                mCachedDataLock;
}
```

These exemplary data structures include the following information. A subject node contains a subject identifier, subject level, pointer to parent channel or subject node, file descriptor for its own directory, pointer to hash table containing its next level subject nodes, and pointer to a data node. A data node contains a pointer to its subject parent node, file descriptor for the data directory, circular buffer containing the data structures for the data stored on each storage device, head and tail of the buffer, and lock for locking the data node during retrieval and storage. The stored time grain node is the node representing the actual data file, and the last time grain node represents the last buffer that has not yet been stored to the storage device but is maintained in memory. The caching and data storage threads in this example use the mutex of the last time grain node for preventing concurrent access to the last time grain node.

Agent Processing

FIG. 16 is a flow chart of an agent method 350 for an outgoing subscription message. Method 350 can be implemented, for example, in software modules as represented by agent 128 for execution by processor 134 in user (subscriber) machine 122. In method 350, agent 128 receives a subscription such as via the method described above in FIGS. 11 and 12 (step 352). Agent 128 creates a string specifying a Boolean expression for the subscription (step 354) and parses the string to detect any errors in the subscription (step 356). If an error exists, agent 128 can present an error message to the user (step 360) in order for the user to correct the error and re-enter the subscription. If the subscription contains no errors (step 358), agent 128 stores the expression in a data structure, an example of which is provided below (step 362). Agent 128 translates constituent not-equal expressions in the data structure to positive form (step 364) and translates the data structure to a corresponding disjunctive normal form (DNF) structure (step 366). Agent 128 also simplifies AND expressions of the DNF structure to contain only range filters and membership tests (step 368).

The DNF is a well-known canonical form in which a Boolean expression is represented as an OR of one or more sub-expressions called disjuncts, each sub-expression being an AND of one or more attribute tests. For example, the Boolean expression (price >= 10 AND (symbol == "LU" OR symbol == "T")) has an equivalent DNF representation of ((price >= 10 AND symbol == "LU") OR (price >= 10 AND symbol == "T")).

The transformation in step 364 involves translating expressions having the "not-equal" operator (represented in an exemplary syntax as !=) into an equivalent "positive" form that specifies all allowed values rather than the one disallowed value. This transformation is performed prior to creation of the DNF, and it is needed because the routers in this example require formulae to be in positive form. For example, the expression (price != 80) can be transformed to the equivalent positive expression (price <= 79 OR price >= 81).

The transformation in step 368 is performed after the DNF is created and involves an extra simplification of the resulting AND expressions, and it is also performed to simplify the work of the routers in this example. In particular, an AND of multiple attribute tests for the same attribute can be simplified into a canonical "range filter" having either one lower bound, one upper bound, both a lower and upper bound, or a single value in the case of an equality test. The particular kind of range filter is then encoded according to Table 22.

For example, the expression (price >= 10 AND price <= 80 AND price >=20 AND price <= 100) can be simplified to the expression (price >= 20 AND price <= 80), which is an example of a range filter with both a lower and an upper bound. Examples of the other kinds after simplification are the following: (price >= 20) (lower bound only); (price <= 80) (upper bound only); and (price == 50) (single value). In creating these range filters, it is possible that some sub-expression will simplify to true or to false, in which case the sub-expression can be eliminated according to the laws of Boolean algebra, thereby further optimizing the encoding of the expression in a message. For example, the expression (price >= 50 AND price <= 20) simplifies to false, since no value for "price" can satisfy the expression. In the special case in which a whole filter expression simplifies to false, the agent need not create a message at all, thereby relieving the router of unnecessary work.

If the subject filter contains wildcards, agent 128 can optionally convert them as explained below (step 370). Otherwise, any wildcards can be converted in the network, rather than on the user machine or other device. In this exemplary embodiment, the syntax for subject filters is the only syntax that uses wildcards, and the syntax for attribute filters is the only syntax that uses Boolean expressions. Alternatively, implementations can use different or varying types of syntax for subject filters and attribute filters.

Agent 128 encodes the resulting DNF expression into a message (step 372) and transfers the message to an intelligent router (step 374). The encoding can involve converting the subscription to a flat message format, meaning that it constitutes a string of data. This transferring can involve propagating routing rules generated from subject filters and attribute filters for the subscription to one or more intelligent routers or other routing entities in the network. For the propagation, the subscription expression can be mapped into a conventional packet structure, for example.

The encoding for step 372 involves marshalling subscriptions for a channel into a messaging format of the messaging API for propagation throughout a channel. A subscription is internally messaged, for example, as a notification with subject #.SUBSCRIPTION. Because there are both a variable number of subject filter fields and a variable number of attribute tests, one pair of bytes is used to store the number of subject filter fields, and another pair of bytes is used to store the number of attribute tests in this example. The individual fields of the subject filter are marshaled sequentially, for example, in the order in which they were specified in the original subscription and are each marshaled into a two-byte portion of the message. Wildcard fields can be marshaled as described below.

In marshaling the attribute tests, the operands of the tests are marshaled at the end of the message in a manner similar to the marshaling of attribute values of notifications. Prior to marshaling the attribute tests and operands, they are sorted by attribute order within each disjunct of the DNF with tests on predefined attributes in position order, followed by tests on discretionary attributes in name order. Furthermore, the set of relational tests on scalar valued attributes within each disjunct are simplified to a canonical form as range filters having either one limit (for left- or right-open ranges or equality tests) or two limits (for closed ranges between distinct limits). The remaining information about the tests is encoded into, for example, two-byte pairs in the same order as the operands; this sequence of two-byte pairs is placed in the message immediately following the sequence of two-byte encoding of subject filter fields. The two-byte pairs can constitute one form of a sequence of bit-string encodings of attribute tests, which can also be used to represent other types of encodings aside from two-byte pairs. Examples of attribute tests are provided below.

The schema for the encoding of the attribute tests is depicted in Table 20. Table 21 illustrates encoding for the two-byte pairs, and Table 22 illustrates encoding of the Operator ID in the two-byte pairs.

Table 20	
Encoding Rules	
1	A zero in the D bit indicates the beginning of a new disjunct in the DNF, while a one in the D bit indicates an additional conjunct within the current disjunct.
2	A value other than all ones in the Notification Attribute Position indicates the position of a predefined attribute (as defined by the channel's notification type) to which the test applies; the operand for the test is marshaled as depicted in the example shown in FIG. 18.
3	A value of all ones in the Notification Attribute Position indicates that the test applies to a discretionary attribute, in which case the name length and name of the attribute to which the test applies are marshaled with the operand.
4	The bits for the Operand Type ID encode one of the predefined types for attributes.
5	The bits for the Operator ID encode the operator used in the test, as defined in Table 22.

Table 21							
First Byte							
0	1	2	3	4	5	6	7
D	Notification Attribute Position						
Second Byte							
0	1	2	3	4	5	6	7
Operand Type ID					Operator ID		

Table 22	
Operator	Operator ID
Left-open range	000
Right-open range	001
Closed-range	010
Equality test	011
Positive membership test (in)	100
Negative membership test (not in)	101

Because the two-byte pair for a test already indicates both the type of the operand of the test and whether or not the test applies to a predefined or discretionary attribute, there is no need to separately marshal the number of tests performed on discretionary attributes or their types. This scheme assumes there are no more than 127 predefined attributes in a notification. Alternatively, this design may use more bits to encode attribute tests.

While this marshaling convention orders and groups attribute tests according to the DNF of the attribute filter, an infrastructure element (such as a router) may choose to evaluate the tests in some other order (perhaps according to dynamically derived local data about the probability of success or failure of the different tests) in order to make the overall evaluation of the attribute filter more efficient. The Subscription ID field of the message is a value generated by the agent for uniquely identifying the subscription to the agent's edge router in subsequent requests to modify or unsubscribe the subscription. In particular, a dynamic modification to the attribute filter of a subscription is propagated using the message format shown in the example of FIG. 18, except that the subject is #.RESUBSCRIPTION and the Subscription ID is that of the previously registered subscription being modified. And an unsubscription is propagated using, for example, the message format of FIG. 18 up through the Subscription ID field, with the subject being #.UNSUBSCRIPTION and the Subscription ID being that of the previously registered subscription being unsubscribed.

The following provides an example to illustrate the conversion and encoding by the agent as described above. Consider the following example attribute filter expression: price >= 10 and (symbol == "LU" or (volume >= 1000 and volume <= 10000)). FIG. 19 presents a Unified Modeling Language (UML) diagram 390 depicting the objects used by the agent in step 362 to store the expression. This diagram illustrates an hierarchical relationship for specifying the subscription, which can include variables, constant values, or both. The objects in the diagram can be instances of filter classes depending upon a particular implementation. Each SimpleFilter object depicts the values of attributes used to store information about a corresponding attribute test of the filter expression. In the expression of FIG. 19, an OR filter 396 connects two AND filters 392 and 400. The AND filter 392 contains a simple filter 394 with attributes for the subscription. Likewise, the OR filter 396 contains a simple filter 398, and the AND filter 400 contains simple filters 402 and 404.

For the purposes of this example, attributes price, symbol, and volume are assumed to be predefined attributes of the associated channel and are assumed to be defined in positions 0, 1 and 2, respectively. Furthermore, the types of the attributes are assumed to be unsigned integer (typecode 6), character array (typecode 12), and unsigned integer (typecode 6), respectively.

Consider next a subscription containing the above example attribute filter expression as its attribute filter. FIG. 18 presents the marshaling of the subscription into a message. The schematic 386 on the left side of FIG. 18 shows the actual message contents, while the schematic 388 on the right provides a legend for the different parts of the message. The width of each schematic in this example is four bytes. Prior to marshaling, the filter has been converted to its equivalent DNF: (price >= 10 and symbol == "LU") or (price >= 10 and volume >= 1000 and volume <= 10000).

The sixteen-bit attribute test encodings are shown as bit sequences, with gaps showing the separation into the different parts. Note that the two tests on price in this example cannot be combined since they are in separate disjuncts, and thus they are marshaled separately as ranges that have no right bound ("right-open ranges"). On the other hand, the two tests on volume can be combined since they are in the same disjunct, and thus they are marshaled together as a single "closed-range" test.

Finally, note also that certain fields are characterized as being "assumed"; this means that values for these fields were chosen arbitrarily for this example and are in general independent of the subscription that was marshaled. In addition, the subject filter for the subscription was arbitrarily chosen to be ">," which matches any subject defined by the associated channel. The example described above and shown in FIGS. 18 and 19 is provided for illustrative purposes only, and the marshalling can be used with any other type of subscription. Also, method 350 provides only one example of marshaling subscriptions, and they can be marshaled in any other way.

FIG. 17 is a flow chart of an agent method 376 for an incoming message. Method 376 can be implemented, for example, by agent 128 and application 126 in user machine 122. In method 376, agent 128 receives a message from an intelligent router corresponding with a subscription (step 378). Agent 128 determines a channel corresponding with the subscription (step 380), for example by the channel ID in the message, and calls an API for the channel (step 382). The API present the data for the subscription in a GUI or other format at the user machine (step 384). The processing of incoming messages can use a process of decoding the data in the reverse of the encoding process described above, and this decoding (reverse encoding) can be performed in a router or in other network entities.

Wildcard Processing

FIG. 20 is a flow chart of a wildcard method 410. This method illustrates an example of using a set of routing rules for a filter to convert wildcards in expressions for subscriptions. Method 410 can be implemented, for example, in software modules as represented by agent 128 for execution by processor 134 in user machine 122. Alternatively, wildcards can be processed in the network by processor 93 under software control in intelligent router 92 or in the corresponding functions contained in ASIC 91. Wildcards include open fields or variable length fields, examples of which are provided in Table 21.

In method 410, agent 128 or other entity receives a subscription having a wildcard (step 412). The subject length for subscriptions can be specified by a publisher when publishing content, and the subject can be pre-processed on the publisher machine, for example, to count the fields of the subject and thus obtain a field count (length) for it. Agent 128 counts the number of fields in the filter operand (step 414) and initializes a new rule (filter) of field length = N (step 416). Agent 128 retrieves a sub-field for the subscription (step 418) and determines if the filter operand sub-field O[i] is a wildcard (step 420). If the filter operand sub-field is not a wildcard, agent 128 adds a conjunctive clause to the rule, field [i] = O[i] (step 422). If the filter operand has more sub-fields (step 424), agent 128 returns to step 418 to process additional sub-fields. The parameter "i" represents a field where i is an integer representing the field number in this example.

After processing the sub-fields, agent 128 determines if the last filter operand sub-field is a ">" (step 426) and, if so, it changes the length constraint to field length > N-1 (step 428). Wildcard processing can use any type of symbol, and a ">" is only one such example. In this example, a "a.>" can mean a.b, a.c, a.d, etc. and all their sub-subjects at all levels (for example, a.b.x, a.c.x, a.b.x.y, etc.). Other symbols can be used for other implementations of wildcards.

If necessary, agent 128 propagates the transformed rule to intelligent routers or other entities in the network (step 430). Accordingly, the method iterates through the sub-fields in order to process them for conversion of the wildcards to non-wildcard rules, meaning rules that do not contain wildcards. The conversion of wildcards can occur anywhere in the network, for example on the subscriber machine or in an intelligent router. The conversion can thus occur in one entity with the transformed rule propagated to other entities or it can occur dynamically.

Table 23 provides a summary, along with examples, of these exemplary routing rules for processing wildcards. These routing rules can be generated in the intelligent routers, for example, or generated in other network entities and propagated to the intelligent routers. In addition, the routing rules in Table 23 are provided for illustrative purposes only and other routing rules are possible for converting wildcards.

Table 23

Original Rule	Transformed Rule
subject = “a.b”	subject.length == 2 & subject[0] == “a” & subject[1] == “b”
subject = “C.*.D”	subject.length == 3 & subject[0] == “C” & subject[2] == “D”
subject = “foo.>”	subject.length > 1 & subject[0] == “foo”
subject = “*.*.b.*.c.>”	subject.length > 5 & subject[2] == “b” & subject[4] == “c”

Caching with Selective Multicasting

Message persistence is the ability to store messages and retrieve them at a later time. A large number of specific applications, *e.g.*, email, generally require lengthy message persistence for messages flowing through the network. In ideal conditions, with no failures in the network an always-connected subscriber should not need any persistence beyond that required for these specific applications. However, in reality, messages can get “lost” while traversing through the network due to various reasons – *e.g.*, (1) failures or buffer overflows occurring either inside the network or at the user end or (2) users doing an explicit disconnect from the network and connecting back again after a time period.

The persistence model of the event notification system described herein is divided into two levels: short-term persistence and long-term persistence. Short-term persistence is designed for recovering from packet lost due to network congestion or short-term link failure. Long-term persistence is designed for recovering from other failures including, *e.g.*, the loss of user connections or ISP network failure, failure of user machines, longer-term network failure, and/or other failures. Embodiments of these two schemes are described below.

Short-term persistence: Data Retransmission and Flow control

In a data network, a cause of data loss can be simply classified as link failure and buffer overflow. To provide reliable channels for the event notification system, these issues need to be addressed. For link failures, it is possible to enforce a forward error correction (FEC) scheme to correct some errors caused by link failures. However, it is still necessary to provide a scheme to recover packets when the error is so serious that no FEC scheme can correct it. As for buffer overflow, it is necessary to prevent the buffer flow from happening. Flow control schemes are typically used in data network to avoid such problems.

In the short-term persistence scheme, Transmission Control Protocol (TCP) tunnels are preferably used to connect event routers (*e.g.*, intelligent routers 12) hop-by-hop. Reasons for relying on a reliable layer-2 tunnel instead of using a reliable transport protocol (*e*.g., RMTP) are multi-fold. In a short-term persistence scheme in the event notification system, messages are preferably filtered out by routers if the messages do not satisfy the filter rules. Consequently, a receiving router generally can not detect the loss of packets by using schemes like source sequence number. Likewise, it is also not desirable for all receiving routers to acknowledge on each packet they receive because such this would cause an overload of acknowledgements (*i.e.*, an ACK-explosion). Besides, to avoid the buffer overflow, to the short-term persistence model implements a flow control scheme so that before a router runs out of buffer space, the router can request a neighboring router forwarding messages to it to slow down. These schemes are covered by TCP.

TCP transmission policy: In TCP used for the short-term persistence scheme, a transmission window is preferably used locally for the data sender to help keep track of data that has been retransmitted. The purpose of using transmission window is two-fold: first of all, the transmission window ensures that the sender will know explicitly that the data has received by the receiver correctly; secondly, the transmission window allows better usage of the channel capacity. In TCP, each byte sender sent is required to be acknowledged, implicitly or explicitly. The transmission window helps the sender to keep track of data that has been sent and acknowledged. The transmission window also improves the channel utilization as a sender is allowed to send data within the transmission window rather than having to stop and wait for the previous packets to get acknowledged. Once previous data is acknowledged, the window will be automatically advanced.

A receiver window is also maintained in TCP. The receiver window is preferably used to indicate the available buffer space at the data receiver end. it's the available buffer space value is sent to the sender so that the sender knows how to avoid overflow the buffer at the receiver side.

TCP congestion control: Since TCP is designed as an end-to-end transport protocol, the TCP utilized in the short-term persistence scheme also addresses buffer overflow inside the publish-subscribe network. To address this, TCP used for the short-term persistence model preferably uses a third window: the congestion window. The congestion window is used for the sender to guess the maximum buffer space on the routers along the path. In short, the congestion window size is reduced if the sender detects a loss in packets, or is increased vice versa.

Long-term Persistence: Caching for Persistent Channels

A channel (*e.g.*, as described above) can either be persistent or real-time. A real-time channel transmits data that is generally only useful in real-time and does not have any application-specific persistence requirements. A persistent channel stores data traversing through network for a persistence time frame T. In other words, persistence for a persistent channel is guaranteed for a time frame T. This persistence of data is achieved through the following, for example: caching data at each edge node for the persistent duration of a channel; retrieving data from the cache transparent to the users under failure conditions; allowing the user to explicitly retrieve data from the cache; making the flow of data through the network persistent by guarding against router failures and setting up reliable tunnels between routers; and, protecting the channel components against failure through replication.

Therefore, as described below, the long-term persistence scheme preferably enables a subscriber registered with a persistent channel to retrieve the old data cached in the network for the last "X" timeframe (X < T), when the subscriber crashes and comes back up again within the time frame T for the persistent channel.

In the long-term persistence scheme, subscriber applications (*e.g.*, application 126) preferably can explicitly pull data (*e.g.*, messages) from an associated subscriber agent (*e.g.*, agent 128). As described above, agents can make use of or be implemented with proxies. After the agent, or proxy, has recovered from a network failure, the agent preferably transparently retrieves data from the cache for the duration that it was disconnected from the edge router. Also, a subscriber is preferably allowed to access only data up to last T time frame in the long-term persistence scheme. To this end, time is preferably determined with respect to the edge router to which the agent (or proxy) is connected. Retrieved cached data is preferably delivered out of band and with no real-time guarantees. The embodiment of the long-term persistence scheme is targeted towards an already existing subscriber who crashes and comes back up again or loses connection with an edge router (*e.g.*, edge router 16). A new subscriber may not be able to get cached information.

Definition of Persistence: Timed Persistence (with time frame T) to a subscriber is defined as the ability to retrieve the last time frame T of data from the publish-subscribe network. If the subscriber leaves the network, any data on a persistent channel that is received during the subscriber's absence is held in the network for a time frame T (from the data's receipt). If the subscriber returns within the timeframe T, the subscriber does not lose any data. However, if the subscriber returns between T and 2T timeframe, the subscriber may loose data. If the subscriber returns after the timeframe 2T, the subscriber is preferably not guaranteed access to any previous data.

The above definition requires that the publish-subscribe network tree leafed at the subscriber should be retained for time frame T after the subscriber disappears and then can be pruned, so that new data is received for the time frame T after the subscriber goes away is retained until the time frame T reaches its expiry time.

Architecture: FIG. 21 is a block diagram illustrating certain components of a publish-subscribe network that provide persistence through caching. As shown, the network includes core routing nodes 548 and an edge routing node 545. Each routing node preferably includes an intelligent router 92 (shown with the edge routing node) and a conventional backbone router (not shown), as described above in FIG. 4. Each intelligent router 92 that needs to perform caching for persistent channels preferably has a cache manager 218 co-located with it, as illustrated by FIG. 21. The cache manager 218 is described above with references to FIG. 8. The intelligent router 92 is preferably responsible for short-term persistence for retrieving lost data or recovering from router failures. The cache manager 218 is responsible for caching data to provide long-term persistence for a channel. The cache manager 218 preferably caches this data in the cache 540. The cache 540 preferably includes a memory and a disk (not shown).

There are several advantages to having the cache manager 218, as opposed to the intelligent router 92, responsible for caching data for long-term persistence, including: the compute-intensive operation of indexing the cached data can be performed on a separate processor, so the performance of the routing and filtering processor is not affected and disk I/O operations for periodically moving cached data to disk can also be done on another processor thus preventing cycles from being stolen from routing and filtering and sparing the edge router from having to do regular I/O.

Also shown in FIG. 21 is an agent 128, which is preferably resident in subscriber machine 122 (not shown in FIG. 21), as described above in FIG. 5. The agent 128 is responsible for communicating with the cache manager 218 to retrieve data from the cache 540, receiving the retrieved data and for organizing the retrieved data. As noted abov.e, the agent 128 can make use of or be implemented with a proxy.

Under no failure conditions, only edge router nodes 545 need to have a cache manager 218 associated with them. However, although not shown in FIG. 21, since the long-term persistence scheme anticipates failures, each of the first level of core routing nodes 548 upstream from the edge routing node 545 preferably includes a cache manager 218 that stores data. Upstream is the direction moving away from the agent 128 (*i.e.*, away from the subscriber machine 122). The first level of upstream core routing nodes refers to the routing nodes immediately upstream from the edge routing node 545. Although publish-subscribe networks often include a plurality of first level upstream core routing nodes, FIG. 21 only depicts one first level upstream core routing node, core routing node 548. As described above, a cache manager 218 provides for local caching of data at a network node at which it is located. Therefore, the operation of cache managers 218 located at various core routing nodes, including, *e.g.*, core routing node 548, provides for distributed caching of data throughout the network core. This distributed caching provides a backup for the caching at edge routing node 545.

FIG. 22 is a diagram illustrating backup caching in an upstream router (*e.g.*, core routing node 548). In the long-term persistence scheme, each cache is preferably backed up by the next upstream router's cache. An upstream cache stores all incoming data and acts as a backup for all next level downstream edge router caches. The data in upstream caches is preferably stored using the same mechanism as the edge router cache.

With reference now to FIG. 23, the architecture for caching for persistent channels preferably provides functionality spanning across four different modules: the cache manager 218 – preferably a server process responsible for storing data going through the intelligent router 92; router cache API 552 – preferably a library responsible for all control plane accesses to the cache manager 218 from the intelligent router 92, *e.g.*, creating and destroying the cache; agent (or proxy) cache API 554 – preferably a library responsible for all control plane accesses to the cache manager 218 from the agent 128 (or agent 128 proxy), *e.g.*, retrieving data; and, the agent 128 (or proxy) – preferably responsible for collecting retrieved data from the cache 546 and organizing the data.

FIG. 23 illustrates the interaction of these four modules. Both the agent 128 and the intelligent router 92 preferably access the cache through the cache API libraries 552 and 554. The cache API libraries 552 and 554 provide API for initializing into the cache 546, creating and destroying caches for subject, retrieving cache addresses and, most importantly, retrieving the data from the cache 546. The routing daemon 216 preferably sends data to the cache manager 218 through the data path without going through the cache API 552. The cache APIs 552 and 554 preferably use the control path for all control messages including data retrieval.

Cache Manager - Cache Management: With reference now to FIG. 24, when the cache manager 218 encounters a new channel, the cache manager 218 preferably invokes an information server (*e.g.*, servers 152, 154 and/or 156 described above) to get the channel manager 150 for the channel. Once the cache manager 218 has the channel manager's 150 address, the cache manager 218 preferably retrieves the channel properties from the channel manager 150. The channel properties preferably include, for example: channel subject tree and attributes, persistent properties of the channel, persistent time frame (T) for the channel, granularity of caching. Before the cache manager 218 can start caching the data flowing through a channel for a given subject, that subject's cache needs to be created in the cache manager 218. The cache manager 218 expects a create cache message and in response to the message creates the subject cache. This subject cache can then be destroyed, suspended or resumed on request. FIG. 24 illustrates cache creation on subscription.

Cache Manager - Cache Data Input: The cache manager 218 preferably has access to the data coming into the intelligent router 92 in a number of ways, for example: an IP like solution in which case all the data on the intelligent router's 92 incoming link is also forwarded to the cache manager 218; using a sniffing mechanism (in which the cache manager 218 listens to all packets traveling on the network of the intelligent router 92); after filtering, the intelligent router 92 forwards each message, that needs to be propagated on one or more links, to the cache 546; and, the cache manager 218 acts as a subscriber for all data coming into the intelligent router 92.

Cache Manager - Cache Data Storage: With reference now to FIG. 25, the cache manager 218 preferably indexes the data in the cache 546 in a number of ways, *e.g.*, channel id, subjects, publisher id, timestamp, time grains (G), primary caching attribute, link (in the special case when caching is done for failures) or other ways. The data may be indexed and stored in a hierarchical directory structure in the file system or in memory. The data preferably is cached in memory and periodically moved to disk. The caching in memory is only for the duration of the "G" time grains. After the time G is expired all the data related to a particular branch in the tree is preferably moved to the file under that branch overwriting the earliest file for that branch. (Note that the G preferably is not implemented as a sliding window, but as an absolute window, because it is expensive to write each message to disk individually, and more efficient to write all of the G interval to the disk in one operation). FIG. 25 is a diagram of an exemplary indexing tree. When caching data for persistence the first indexing tree in FIG. 25 is preferably used.

With continued reference to FIG. 25, the subjects preferably are stored in a hierarchy, where "a" is the parent of subjects like "a.b", "a.c" "a.d", etc. The cache manager 218 preferably keeps a hash table for the cache 546 mapping all subjects to their corresponding file locations. In some cases, the cache 546 may need to store data under failure conditions when an upstream router (*e.g.*, core routing node 548) detects the failure of a downstream router (*e.g.*, edge routing node 545) on one of its links. The first approach for recovery is to restart the downstream router (which could take minutes). While the downstream router is being restarted, the upstream router will need to cache the data that is being forwarded down that link. This cache (*e.g.*, called the FM Cache in FIG. 25) is preferably indexed on outgoing links.

Cache Manager - Garbage Collection: If a channel is not persistent, the cache 546 does not store the data, but drops it immediately. If the channel is persistent, the cache

546 stores the data. A persistent time frame "T" for a particular channel is divided into N time grains each of size G. The caching in memory is only for the duration of G. After the cache manager determines that time interval G has passed, the data is moved to the disk. The cache manager 218 stores the data on the disk for the duration of persistent time frame interval T.

The data corresponding to a time interval G is deleted from the disk once the time becomes greater than the Persistent timeout (T) for the channel + the upper limit of the interval. To better understand this, suppose a channel has a T of 2 hours. As an example, the cache manager 218 uses a time granularity G of 15 minutes. For deleting the data from the disk, the policy preferably used is that when the last data cached during a time interval G (15 minutes) has been stored for T (2 hours), the entire data cached during that 15-minute interval will be discarded. Therefore, the data cached in the beginning of that 15-minute interval will have been stored for longer than 2 hours before it is deleted. In this example, the data cached during each 15-minute interval is a block of data. If the persistent time frame T is divided into N intervals, any point in time there will be N+1 blocks of data (N on disk and 1 in memory) in the cache 546 for each subject.

Cache Manager - Cache Data Retrieval: With reference now to FIG. 26, the agent 128 (or proxy) preferably invokes a GetCache operation to get the data going back time "T" from the current time. The cache manager 218 to which the agent 128 connects to invoke the GetCache operation is labeled the Portal Cache in FIG. 26. Due to failure/disconnects of routers or the agent 128, the portal cache may not have all the data requested by the agent 128. In this case, it is the job of the portal cache to retrieve data from all the other caches (*e.g.*, upstream caches), collate the data and return it to the agent 128. FIG. 26 illustrates retrieval from multiple caches (A, B and C) for different time stamps (TS1, TS2 and TS3).

The cache manager 218 preferably can only retrieve data in blocks of time grain G. So the agent 128 may get more data than it expects or requests. In addition, during retrieval from multiple caches, there maybe some overlapping intervals between the caches, so the agent 128 will also see duplicates of data and the agent 128 should do duplicate suppression on the data stream provided by the cache.

Interaction of Cache Manager with other modules: The cache manager 218 preferably interacts with several modules in the event notification system infrastructure, as shown in FIG. 27. When the cache manager 218 encounters a new channel (at create cache time), it preferably invokes the Information Server 550 to get the channel manager 150 for the channel. Once the cache manager 218 has the channel manager's 150 address, it preferably gets the channel properties from the channel manager 150. An administrator module 552 is preferably allowed to set/modify some properties, like the granularity of caching. The administrator module 552 is preferably also allowed to manually create or delete channel cache.

<u>Agent Cache API - Application – Agent Interaction</u>: The application (*e.g.*, application 126) preferably invokes the agent cache API 554 to get the cache 546 with a given subject and filter. Preferably, an application can only retrieve data from the cache 546 if it has already subscribed to that data. The agent cache API 554 preferably actually provides two APIs.

The first API allows an un-subscribed application to subscribe and retrieve a cache 546 at the same time. If a "fifo" flag is set, the subscription is created and sent to the edge router node 545. However, the subscription preferably is immediately put in a "pause" mode. After the agent 128 has received all cached data, the agent 128 first delivers all the cached data, keeps track of the last sequence not seen for all publishers in the data and then delivers the paused data from the last sequence not seen for each publishers.

For the second API, it is assumed that the application has already subscribed to some data and is asking for cached data. In this case, the application has already been delivered some data which cannot be sequenced with respect to the cache data. Hence the "fifo" flag in this case just indicates that the data retrieved from the cache 546 should be sequenced within itself, but need not be sequenced across the regular data stream.

The agent 128 preferably retrieves all the events in one big block of data. After retrieving the data from the Cache API 554, the agent 128 preferably needs to do the following operations on the data before invoking a callback operation (see above): construct notifications from the list of notifications; keep track of the last sequence number for each publisher; and, filtering. When the agent 128 is done pushing all the events to the callback, it preferably sends a DoneCache event to the callback, to indicate that all cached data has been delivered. At this point, if the subscription is FIFO and the regular data is paused, the agent 128 preferably forwards all the paused notifications. The agent 128 delivers only those notifications whose sequence numbers are greater than the last sequence number in the cached data.

Agent Cache API to Cache Interaction: When the subscriber asks for the cached data, the cache API at the agent 128 end 554 preferably first looks up the history of edge routers that the agent 128 was connected to and filters the list using the time interval provided in the GetCache request. The API 554 then sends a GetCache(channel, subject, filters, local_pubs, time_period, fifo, array of routers) message to the last edge cache it was connected to. The cache manager 218 preferably pulls out the data based on the channel id, subject and timestamp and pushes out the data back to the agent 128. When the cache manager 218 is done pushing out the data it sends a DoneCache event to the Cache-API to indicate that the data transfer has completed.

If the cache manager 218 does not find the data locally, it uses the "list of routers" provided by the agent 128 to locate the data needed. Once the cache manager 218 has collected all the necessary data, the cache manager 218 collates the necessary data and does duplicate suppression on it before forwarding it to the agent 128.

Cache Connection History: In order to be able to retrieve data from the caches 546, the cache connection history, for both edge as well as upstream caches, is preferably maintained at the agent 128. Since this information is needed across agent 128 shutdowns and crashes, the information should be maintained persistently in a file. Cache connection history on the disk is preferably stored in the following files and format:

Edge cache locations: The location of the edge cache (*e.g.*, the cache 546 at the edge routing node 546) is preferably obtained from the channel manager/channel library. This occurs at boot-up time and any subsequent time that the edge cache changes, *e.g.*, lost/regained connection, moved connection. The dispatcher notifies the agent 128 of any changes in the edge cache connection and these changes are then communicated to the agent 128 cache library. Each time a change occurs, it is made persistent.

Persistent Storage: CACHE_ROOT/channel_id/Channel – an exemplary path for the cached data.

The data is preferably stored in the following format:

Number of Edge Caches;

Edge Cache1: Number of time intervals, StartTime1:EndTime1, StartTime2:EndTime2,...; and

Edge Cache2: Number of time intervals, StartTime1:EndTime1, StartTime2:EndTime2,...

...with the latest timestamp being the first in the list. Note that the two different edge caches will never have an overlapping interval (because the agent 128 is connected to only one edge cache at a time). Each time a new entry is added, the old entries are checked to see if they are still valid; if they are invalid, the entry is thrown out. A time interval becomes invalid if

Interval EndTime + channel's persistent timeout < current time.

An edge cache entry becomes invalid if all the intervals in the entry are invalid. Note that an "EndTime" of 0 means that the interval is currently active.

Upstream Cache locations: The location of the upstream caches (*e.g.*, the cache 546 at the core routing node 548) is dependent on the subject. Each subject has its own multicast tree and hence the set of first level upstream caches is a function of the subject. Any time that the user subscribes to a subject, the intelligent router 92 preferably returns the list of upstream caches associated with the subject. Similarly, any changes in the upstream cache locations due to failures or reorganizations in the multicast tree are preferably also communicated to the agent 128 through the control channel. These changes are documented locally in a persistent store (file).

Persistent Storage: CACHE_ROOT/channel_identifier/subject (not in a hierarchy, but a full subject). – an exemplary path for the cached data.

The data is preferably stored in the following format:

Number of Upstream Caches;

Upstream Cache1: Number of time intervals, StartTime1:EndTime1, StartTime2:EndTime2,...;

Upstream Cache2: Number of time intervals, StartTime1:EndTime1, StartTime2:EndTime2,...

...again, with the latest timestamp being the first in the list. Unlike the edge cache intervals, two upstream caches can have overlapping intervals, because an agent 128 can have several upstream caches for a given subject. The contents of the upstream cache files are also garbage collected using the same algorithm as the edge caches.

Cache validity during data retrieval: During the lifetime of the agent 128, it goes through connections to different edge routers and the upstream routers. The agent cache API 554 preferably stores this connection history in the local store. When the agent 128 needs to retrieve last T intervals of data from the cache 546, the agent cache API 554 preferably looks through the connection history to determine the caches to access the data from. The algorithm preferably used for this is as follows: 1) the cache library explores all the edge cache intervals and checks for intervals that fall within the T timeframe. If an interval falls within that time frame, it is added to the list L_e of valid edge caches; 2) the list L_e is sorted using the interval start times; 3) for each interval that is not covered by the edge caches in L_e, explore the upstream caches to get all upstream cache intervals that can cover this interval and add valid intervals to List $L_{u.}$; append L_u to L_e and sort L_e using interval start times, to create L.

This algorithm gives the list of caches L and for each cache the time interval for which to retrieve the data. This list of caches L is then 4) marshaled into a get cache message and sent to the cache manager 218. At the cache manager 218 end, the cache manager 218 preferably 5) un-marshals the cache intervals from the get cache message and recreates the list L sorted in the increasing order of start times. For each interval in the list L: the cache manager 218 preferably 6) checks to see if there is a gap between the previous interval and the current intervals and if there is a gap, asks the local cache for the data. If there is no gap, the cache manager 218 preferably 7) talks to the relevant cache to get the data. The cache manager 218 preferably 8) collates data from all the caches and sends it to the agent 128.

Router Cache API: The router cache API 552 at the intelligent router 92 is responsible for invoking the cache manager 218 to create, destroy, suspend and resume caching for a particular subject. The router cache API 552 also deals with initial configuration - uploading the cache address from the intelligent router 92 to the channel manager 150, so that the agent 128 side (the agent cache API 554) can obtain this information when needed – and retrieving the location of caches 546 for other routers (this is used when the intelligent router 92 wants to notify the agent 128 of the upstream caches for a given subject, *e*.g., in subscription reply and after subject tree changes).

Use of Cache for Pull: The discussion above focuses on the use of cache for implementing persistent channels and allowing returning subscribers to pull data from the network. An alternative embodiment allows any subscriber (new or returning) to pull any kind of data from the caches 546 (*e.g.*, including data that the subscriber may not have already subscribed to, but is in the caches 546 because of someone else's subscription). The difference between this embodiment and the preceding embodiment is that for a returning subscriber the data is guaranteed to the present and the location of the data is well known, while for a new subscriber the location of the stored data is not known. A simple way of implementing this alternative embodiment is to publish a "FindCache" request on the channel. A "FindCache" request contains a channel id, a subject, filters, time interval, and the location of the agent 128 looking for the cache 546 with the requested cached data. All caches 546 listen for the "FindCache" Request. When each cache 546 receives the request, the cache 546 looks to see if the corresponding data is in its datastore and if so, sends back its own location in a unicast message. The agent 128 chooses one of the caches 546 and invokes a GetCache operation on it to get the data.

Last Data Pull: Other embodiments include a feature, the last data pull, that allows a subscriber application (*e.g.*, application 126) to get last message for a given subject. This is useful for data such as stock quote alerts, for example, where the user just wants to know the last stock price and not the history.

Implementation of the Cache Manager: There are preferably three types of threads, for example, in the cache manager 218 implementation: a Data Caching thread – the data caching thread preferably picks up data from the connection to the intelligent router 92 and indexes and stores the data in memory; a Data Storage thread – once the end of a time interval is reached, the data storage thread preferably moves the data stored in memory to disk and in the process also performs garbage collection on expired data; and, a Data Retrieval thread – the data retrieval thread preferably is responsible for picking up requests for cached data and retrieving data from the cache 546. These three types of threads may be implemented as a single thread or a pool of threads. Preferably, the Data Caching thread and the Data storage thread are synchronized during the time that data is being moved to disk. This synchronization between the data storage thread and the data retrieval thread prevents data from being removed while the data is being retrieved.

Data Structures: Examples of data structures for caching are provided above in Table 19 and the accompanying description.

Data Storage: FIG. 28 is a diagram illustrating a preferred directory structure used for storing data files in a cache 546 named "Aquila Cache." Note that each subject level directory preferably has a set of child subject directories as well as a data directory that stores data published on that subject. For example, data published on Fox.Movies on the Entertainment channel will go into a file in the directory AquilaCache/Entertainment/Fox/ Movies/Data and data published on subject Fox will go into a file in the directory AquilaCache/Entertainment/Fox/Data. In order to speed up data storage, the directory hierarchy for a particular subject is preferably created at the time when the intelligent router 92 asks the cache 546 to start caching for a given channel and subject.

Data Retrieval: Data retrieval from the cache 546 should be efficient, so as not to block data storage and hold up the cache 546. The data retrieval retrieves data from both the disk and memory. The steps taken for data retrieval preferably are as follows: 1) locate the data node; 2) lock the data node; 3) locate the time stamps of the data that needs to be retrieved; 3) retrieve and store the data into memory; 4) unlock the data node; 5) filter and sequence the retrieved data stored in memory before pushing it out to the agent 128 client.

While the present invention has been described in connection with an exemplary embodiment, it will be understood that many modifications will be readily apparent to those skilled in the art, and this application is intended to cover any adaptations or variations thereof. For example, various types of publisher machines, user or subscriber machines, channels and configurations of them, and hardware and software implementations of the content-based routing and other functions may be used without departing from the scope of the invention. This invention should be limited only by the claims and equivalents thereof.